1 Introduction

1.1 Python for Scientists

The title of this book is *Python for Scientists*, but what does that mean? The dictionary defines “Python” as either (a) a nonvenomous snake from Asia or Saharan Africa or (b) a computer programming language, and it is the second option that is intended here. By “scientist,” we mean anyone who uses quantitative models either to obtain conclusions by processing precollected experimental data or to model potentially observable results from a more abstract theory, and who asks “what if?” What if I analyze the data in a different way? What if I change the model?

Given the steady progress in the development of evermore complex experiments that explore the inner workings of nature and generate vast amounts of data, as well as the necessity to describe these observations with complex (nonlinear) theoretical models, the use of computers to answer these questions is mandatory. Luckily, advances in computer hardware and software development mean that immense amounts of data or complex models can be processed at increasingly rapid speeds. It might seem a given that suitable software will also be available so that the “what if” questions can be answered readily. However, this turns out not always to be the case. A quick pragmatic reason is that while there is a huge market for hardware improvements, scientists form a very small fraction of it and so there is little financial incentive to improve scientific software. But for scientists, specialized, yet versatile, software tools are key to unraveling complex problems.

1.2 Scientific Software

Before we discuss what types of scientific software are available, it is important to note that all computer software comes in one of two types: proprietary or open-source. **Proprietary software** is supplied by a commercial firm. Such organizations have both to pay wages and taxes and to provide a return for their shareholders. Therefore, they have to charge real money for their products, and, in order to protect their assets from their competitors, they do not tell the customer how their software works. Thus the end users have little chance of being able to adapt or optimize the product for their own use.

Since wages and taxes are recurrent expenditures, the company needs to issue frequent charged-for updates and improvements (the *Danegeld effect*).

Open-source software, on the other hand, is available for free. It is usually developed by computer-literate individuals, often working for universities or similar organizations, who provide the service for their colleagues. It is distributed subject to anti-copyright licenses, which give nobody the right to copyright it or to use it for commercial gain. Conventional economics might suggest that the gamut of open-source software should be inferior to its proprietary counterpart, or else the commercial organizations would lose their market. As we shall see, this is not necessarily the case.

Next we need to differentiate between two different types of scientific software. The easiest approach to extracting insight from data or modeling observations utilizes prebuilt software tools, which we refer to as "**scientific software tools**." Proprietary examples include software tools and packages like Matlab, Mathematica, IDL, Tableau, or even Excel and open-source equivalents like R, Octave, SciLab, and LibreOffice. Some of these tools provide graphical user interfaces (GUIs) enabling the user to interact with the software in an efficient and intuitive way. Typically, such tools work well for standard tasks, but they do offer only a limited degree of flexibility, making it hard if not impossible to adapt these packages to solve some task they were not designed for. Other software tools provide more flexibility through their own idiosyncratic programming language in which problems are entered into a user interface. After a coherent group of statements, often just an individual statement, has been typed, the software writes equivalent core language code and compiles it on the fly. Thus errors and/or results can be reported back to the user immediately. Such tools are called "interpreters" as they interpret code on the fly, thus offering a higher degree of flexibility compared to software tools with shiny GUIs.

On a more basic level, the aforementioned software tools are implemented in a **programming language**, which is a somewhat limited subset of human language in which sequences of instructions are written, usually by humans, to be read and understood by computers. The most common languages are capable of expressing very sophisticated mathematical concepts, albeit often with a steep learning curve. Although a myriad of programming languages exist, only a handful have been widely accepted and adopted for scientific applications. Historically, this includes C and Fortran, as well as their descendants. In the case of these so-called **compiled languages**, compilers translate code written by humans into machine code that can be optimized for speed and then processed. As such, they are rather like Formula 1 racing cars. The best of them are capable of breathtakingly fast performance, but driving them is not intuitive and requires a great deal of training and experience. This experience is additionally complicated by the fact that compilers for the same language are not necessarily compatible and need to be supplemented by large libraries to provide functionality for seemingly basic functionality.

Since all scientific software tools are built upon compiled programming languages, why not simply write your own tools? Well, a racing car is not usually the best choice for a trip to the supermarket, where speed is not of paramount importance. Similarly, compiled languages are not always ideal for quickly trying out new ideas or writing short scripts to support you in your daily work. Thus, for the intended readers of this book, the direct use of compilers is likely to be unattractive, unless their use is mandatory. We therefore look at the other type of programming language, the so-called **interpreted languages**, which include the previously mentioned scientific tools based on interpreters. Interpreted languages lack the speed of compiled languages, but they typically are much more intuitive and easier to learn.

Let us summarize our position. There are prebuilt software tools, some of which are proprietary and some of which are open-source software, that provide various degrees of flexibility (interpreters typically offer more flexibility than tools that feature GUIs) and usually focus on specific tasks. On a more basic level, there are traditional compiled languages for numerics that are very general, very fast, rather difficult to learn, and do not interact readily with graphical or algebraic processes. Finally, there are interpreted languages that are typically much easier to learn than compiled languages and offer a large degree of flexibility but are less performant.

So, what properties should an ideal scientific software have? A short list might contain:

- a mature programming language that is both easy to understand and has extensive expressive ability,
- integration of algebraic, numerical, and graphical functions, and the option to import functionality from an almost endless list of supplemental libraries,
- the ability to generate numerical algorithms running with speeds within an order of magnitude of the fastest of those generated by compiled languages,
- a user interface with adequate on-line help and decent documentation,
- an extensive range of textbooks from which the curious reader can develop greater understanding of the concepts,
- open-source software, freely available,
- implementation on all standard platforms, e.g., Linux/Unix, Mac OS, Windows.
- a concise package, and thus implementable on even modest hardware.

You might have guessed it: we are talking about Python here.

In 1991, Guido van Rossum created Python as an open-source, platform-independent, general purpose programming language. It is basically a very simple language surrounded by an enormous library of add-on packages for almost any use case imaginable. Python is extremely versatile: it can be used to build complex software tools or as a scripting language to quickly get some task done. This versatility has both ensured its adoption by power users and led to the assembly of a large community of developers. These properties make Python a very powerful tool for scientists in their daily work and we hope that this book will help you master this tool.

1.3 About This Book

The purpose of this intentionally short book is to introduce the Python programming language and to provide an overview of scientifically relevant packages and how they can be utilized. This book is written for first-semester students and faculty members, graduate students and emeriti, high-school students and post-docs – or simply for everyone who is interested in using Python for scientific analysis.

However, this book by no means claims to be a complete introduction to Python. We leave the comprehensive treatment of Python and all its details to others who have done this with great success (see, e.g., Lutz, 2013). We have quite deliberately preferred brevity and simplicity over encyclopedic coverage in order to get the inquisitive reader up and running as soon as possible.

Furthermore, this book will not serve as the “Numerical Recipes for Python,” meaning that we will not explain methods and algorithms in detail: we will simply showcase how they can be used and applied to scientific problems. For an in-depth discussion of these algorithms, we refer to the real *Numerical Recipes* – Press et al. (2007) and all following releases that were adapted to different programming languages – as well as other works.

Given the dynamic environment of software development, details on specific packages are best retrieved from online documentation and reference websites. We will provide references, links, and pointers in order to guide interested readers to the appropriate places. In order to enable an easy entry into the world of Python, we provide all code snippets presented in this book in the form of Jupyter Notebooks on the CoCalc cloud computing platform. These Notebooks can be accessed, run, and modified online for a more interactive learning experience.

We aim to leave the reader with a well-founded framework to handle many basic, and not so basic, tasks, as well as the skill set to find their own way in the world of scientific programming and Python.

1.4 References

Print Resources

Lutz, Mark. *Learning Python: Powerful Object-Oriented Programming*. O’Reilly Media, 2013.

Press, William H, et al. *Numerical Recipes: The Art of Scientific Computing*. 3rd ed., Cambridge University Press, 2007.

2 About Python

Python is currently the most popular programming language among scientists and other programmers. There are a number of reasons leading to its popularity and fame, especially among younger researchers. This chapter introduces the Python programming language and provides an overview on how to install and use the language most efficiently.

2.1 What Is Python?

Python is a general-purpose programming language that is extremely versatile and relatively easy to learn. It is considered a high-level programming language, meaning that the user typically will not have to deal with some typical housekeeping tasks when designing code. This is different from other (especially compiled) languages that heavily rely on the user to do these tasks properly. Python is designed in such a way as to help the user to write easily readable code by following simple guidelines. But Python also implements powerful programming paradigms: it can be used as an object-oriented, procedural, and functional programming language, depending on your needs and use case. Thus Python combines the simplicity of a scripting language with advanced concepts that are typically characteristic for compiled languages. Some of these features – which we will introduce in detail in Chapter 3 – include dynamic typing, built-in object types and other tools, automatic memory management and garbage collection, as well as the availability of a plethora of add-on and third-party packages for a wide range of use cases. Despite its apparent simplicity, these features make Python a very competitive, powerful, and flexible programming language.

Most importantly, Python is open-source and as such freely available to everyone. We detail in Section 2.2 how to obtain and install Python on your computer.

Based on various recent reports and statistics, Python is currently the most popular programming language among researchers and professional software developers for a wide range of applications and problems. This popularity largely stems from the ease of learning Python, as well as the availability of a large number of add-on packages that supplement basic Python and provide easy access to tasks that would otherwise be cumbersome to implement.

But there is also a downside: Python is an interpreted language, which makes it slower than compiled languages. However, Python provides some remedies for this issue as we will see in Chapter 9.

For researchers, Python offers a large range of well-maintained open-source packages, many of which are related to or at least based on the SciPy ecosystem. SciPy contains packages for scientific computing, mathematics, and engineering applications. Despite being the backbone of many Python applications, SciPy is completely open-source and funded in some part through NumFocus, a nonprofit organization supporting the development of scientific Python packages. We will get to know some of the packages that are part of the SciPy universe in Chapters 4, 5, and 8.

2.1.1 A Brief History of Python

The Python programming language was conceived by Guido van Rossum, a Dutch computer scientist, in the 1980s. He started the implementation in 1989 as a hobby project over the Christmas holidays. The first release became available in 1991 and Python 1.0 was released in 1994; Python 2.0 became available in 2000. With a growing user base, the development team also started to grow and gradually all the features that we appreciate about this language were implemented. Python 3.0 was released in 2008, which broke the backwards compatibility with Python 2.x due to some design decisions. The existence of two versions of Python that were incompatible with each other generated some confusion, especially with inexperienced users. However, support for Python 2.x ended in 2020, leaving Python 3.x as the only supported version of Python. The example code shown in this book and the accompanying Jupyter Notebooks (see Section 2.4.2) are based on Python version 3.9.12, but this should not matter as future versions should be compatible with that one.

Van Rossum is considered the principal author of Python and has played a central role in its development until 2018. Since 2001, the Python Software Foundation, a nonprofit organization focusing on the development of the core Python distribution, managing intellectual rights, and organizing developer conferences, has played an increasingly important role in the project. Major design decisions within the project are made by a five-person steering council and documented in Python Enhancement Protocols (PEPs). PEPs mainly discuss technical proposals and decisions, but we will briefly look at two PEPs that directly affect users: the Zen of Python (PEP 20, Section 2.1.2) and the Python Style Guide (PEP 8, Section 3.13).

We would also like to note that in 2012, NumFOCUS was founded as a nonprofit organization that supports the development of a wide range of scientific Python packages including, but not limited to, NumPy (see Chapter 4), SciPy (see Chapter 5), Matplotlib (see Chapter 6), SymPy (see Chapter 7), Pandas (see Chapter 8), Project Jupyter, and IPython. The support through NumFOCUS for these projects includes funding that is based on donations to NumFOCUS; for most of these open-source projects, donations are their only source of funding.

One detail we have skipped so far is why Van Rossum named his new programming language after a snake. Well, he did not. Python is actually named after the BBC comedy TV show *Monty Python's Flying Circus*, of which Van Rossum is a huge fan. In case you were wondering, this is also the reason why the words "spam" and "eggs" are oftentimes used as metasyntactic variables in Python example code in a reference to their famous "Spam" sketch from 1970.

2.1.2 The Zen of Python

The Zen of Python is an attempt to summarize Van Rossum's guiding principles for the design of Python into 20 aphorisms, only 19 of which have been written down. These guiding principles are very concise and distill many features of Python into a few words. The Zen of Python is so important that it is actually published (PEP 20) and its content is literally built into the Python language and can be accessed as follows:

```
import this
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one -- and preferably only one -- obvious way to
    do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Please note that these guidelines focus on the design of the Python programming language, not necessarily the design of code written in Python. Nevertheless, you are free to follow these guidelines when writing your own code to create truly *pythonic* code. The term *pythonic* is often used within the Python community to refer to code that follows the guiding principles mentioned here.

These guiding principles are numerous and some of them might not be immediately clear to the reader, especially if you are new to Python programming. We would summarize the most important Python concepts as follows.

Simplicity Simple code is easier to write and read; it improves readability, shareability, and maintainability, and therefore helps you and others in the short term and long term.

Readability It is nice to write code as compact as possible, but if writing compact code requires some tricks that are hard to understand, you might prefer a more extensive implementation that provides better readability. Why? Imagine that your future self tries to modify some code that you wrote years ago. If your code is well-readable, you will probably have fewer problems understanding what the individual lines of code do.

Explicitness We will explain this idea with an example. Consider you are writing code that is able to read data from different file formats. A decision you have to make is the following: will you create a single function that is able to read all the different file formats, or do you create a number of individual functions, each of which is able to read only a single file format? The *pythonic* way would be the latter: each function that you create will explicitly be able to deal with only a single file format in contrast to a single function that implicitly deals with all file formats. Why is this solution favored? Generally, explicit code is easier to understand and less prone to confusion.

Naturally, these concepts are entangled and closely related to each other. However, there is no need to memorize these concepts. You will internalize those concepts that are relevant to you by writing code and reading code written by others. And, of course, nobody can force you to follow these principles in your own coding; but we hope that this section provides you a better understanding of the Python programming language and its design.

2.2 Installing Python

Depending on the operating system you are using, there are several ways to install Python on your computer, some of which are simpler than others. The easiest and at the same time safest way to install Python is to use the Anaconda environment as detailed below.

Alternatively, you can also install Python from scratch on your computer – unless it is already installed. In the latter case, you should be careful not to interfere with the native Python already installed as it might be required by your operating system. This process might be a bit more complicated, but there are detailed installation guides for all operating systems available online. To be on the safe side, we recommend the installation of Anaconda, which comes with Conda, a tool to set up and utilize virtual environments, in order to prevent interference with other versions of Python that might be installed on your computer. Once Python is installed, additional packages can also be installed using Conda and the *Package installer for Python*, pip.

2.2.1 Anaconda and Conda

Anaconda is a Python distribution package for data science and machine learning applications. Despite this specialization, the Anaconda Individual Edition (also known as the "Anaconda Distribution") constitutes a solid basis for any scientific Python installation.

The Anaconda Distribution is provided and maintained by Anaconda Inc. (previously known as Continuum Analytics). Despite being a for-profit company, Anaconda Inc. distributes the Anaconda Individual Edition for free.

Installing Anaconda

Installing Anaconda is simple and straightforward. All that is required is to download the respective Anaconda Individual Edition installer (see Section 2.6) for your operating system and run it. The installer will walk you through the installation process. Note that you will need to agree to the Anaconda license agreement. At the end of the installation routine, you will be asked whether to make Anaconda Python your default Python version, which is a good idea in most cases. If you now start the Python interpreter (see Section 2.4.1), you will be greeted by Anaconda Python. Congratulations, you have successfully installed Anaconda Python on your computer.

Conda

One advantage of using Anaconda is the availability of Conda, an open-source package and environment manager that was originally developed by Anaconda Inc., but has subsequently been released separately under an open-source license. Although, for a beginner, the simple installation process for Anaconda Python is most likely its most important feature, Conda also solves two problems in the background. As a package manager, it allows you to easily install Python packages with a single command on your command line, e.g.,

```
conda install numpy
```

Almost all major Python packages are available through Conda. Packages are available through Conda-Forge (see Section 2.6), a GitHub (see Section 10.3.1) organization that contains repositories of "Conda recipes" for a wide range of packages. Conda-Forge contains more detailed information on how to install packages through Conda, as well as a list of all packages that are available through Conda.

As an environment manager, Conda allows you to define different environments, each of which can have its own Python installation. Although this is an advanced feature and becomes important when you are dealing with specific versions of your Python packages, there is still some benefit for the Python beginner. Some operating systems use native Python installation to run crucial services; meddling with these Python installations can seriously harm your system. By default, Anaconda creates a *base* environment for the user. Since this environment is independent from your system, there is no danger in meddling with your system Python installation. Thus using Anaconda is safer than using your system Python installation.

It is not complicated to define new Conda environments and to switch between them. However, due to the advanced nature of dealing with different environments, we refer to the Conda documentation to learn more about how to do this.

2.2.2 Pip and PyPI

Pretty much all Python packages are registered with the *Python Package Index*, PyPI, which enables the easy distribution of these packages. Installing packages from PyPI is very easy using the pip package manager, which comes with most Python installations, e.g.,

```
pip install numpy
```

Everybody can publish their code via PyPI; in Section 10.3.2 we will show how this can be achieved. Since PyPI is the official repository of Python packages, pretty much all available packages are installable using pip.

Pip or Conda?

After learning about Conda and pip you might be confused which of these tools you should use to install Python packages. The short answer is, in most cases it does not matter. Especially for beginners, it is perfectly fine and typically also safe to install packages using pip. Pip is typically faster than Conda in installing packages.

This faster installation process comes at a (small) price that won't matter to most users. The price is that Conda is generally safer in installing new packages. Before Conda installs a new package, it will check the version numbers of all packages that are already installed in your current Conda environment and it will check whether these packages in the present versions are compatible with the new package and vice versa. Pip simply checks whether the versions of the installed packages are compatible with the new package – and it will update the already present packages, to make them compatible with the new package. However, pip disregards that there might be requirements by other packages that will break by updating these existing packages. As a result, pip may break packages that were previously installed.

This happens very rarely, since most Python packages are compatible over many different versions. However, in the case of quickly developing projects it is mandatory to use specific versions of packages. In those cases, it is much safer to use Conda to install new packages. For most other users, especially on the beginner level, there should be no major issues.